

DBTLAB – Exercise 9: Access Path Selection and Plan Generation

Viktor Rosenfeld

based on material from Volker Markl, Alexander Alexandrov, Jonas Traub, Martin Kiefer

Committee Elections at TU Berlin



WHICH committees are being voted?

Faculty Boards

(Extended) Academic Senate

Board of Trustees

from **30. January - 01. February 2024**
10 - 15 h

WHY should I participate?

Exercise your **right** to vote to help shape our university!

➤ **Indirect influence** on the future of TUB

Working as an elected member of a committee:

➤ **Direct influence** on the future of TUB

- Network with interesting people
- Learn new perspectives
- Gain a wealth of experience of democratic opinion-forming processes

*Students have a right and duty to be involved in all committees!
Their involvement is decided by election*

So: TURN OUT AND VOTE!

further infos: www.tu.berlin/themen/wahlen

WHAT are they responsible for?

Committees decide on:

- **Content** and requirements of degree programs
- **Teaching** workload of academic staff
- **Appointments** of new professors
- and much more besides

HOW TO VOTE?

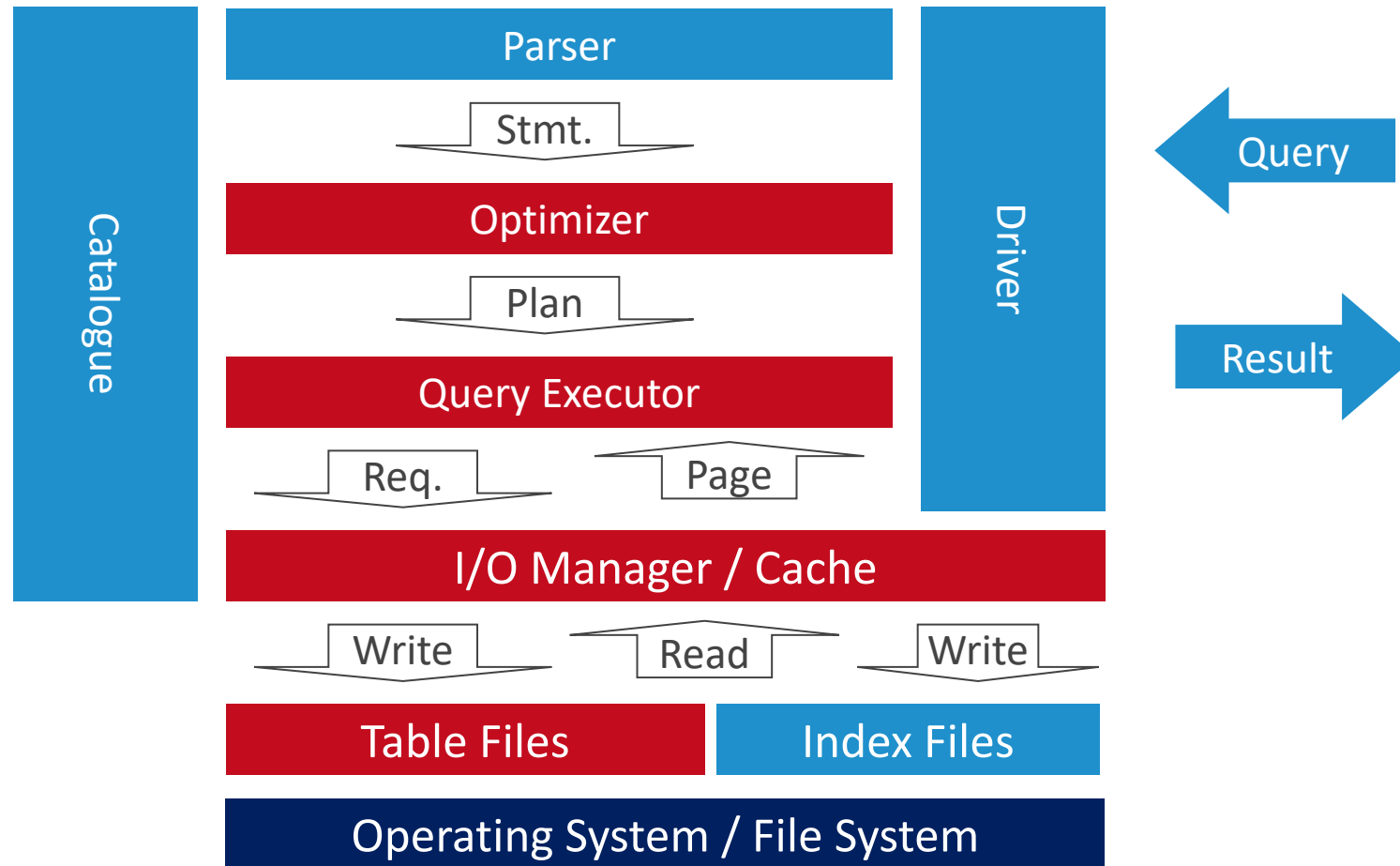
The elections shall be conducted as ballot box elections.

Postal voting can still be submitted until 24.01.24 via personal access in the TU portal.

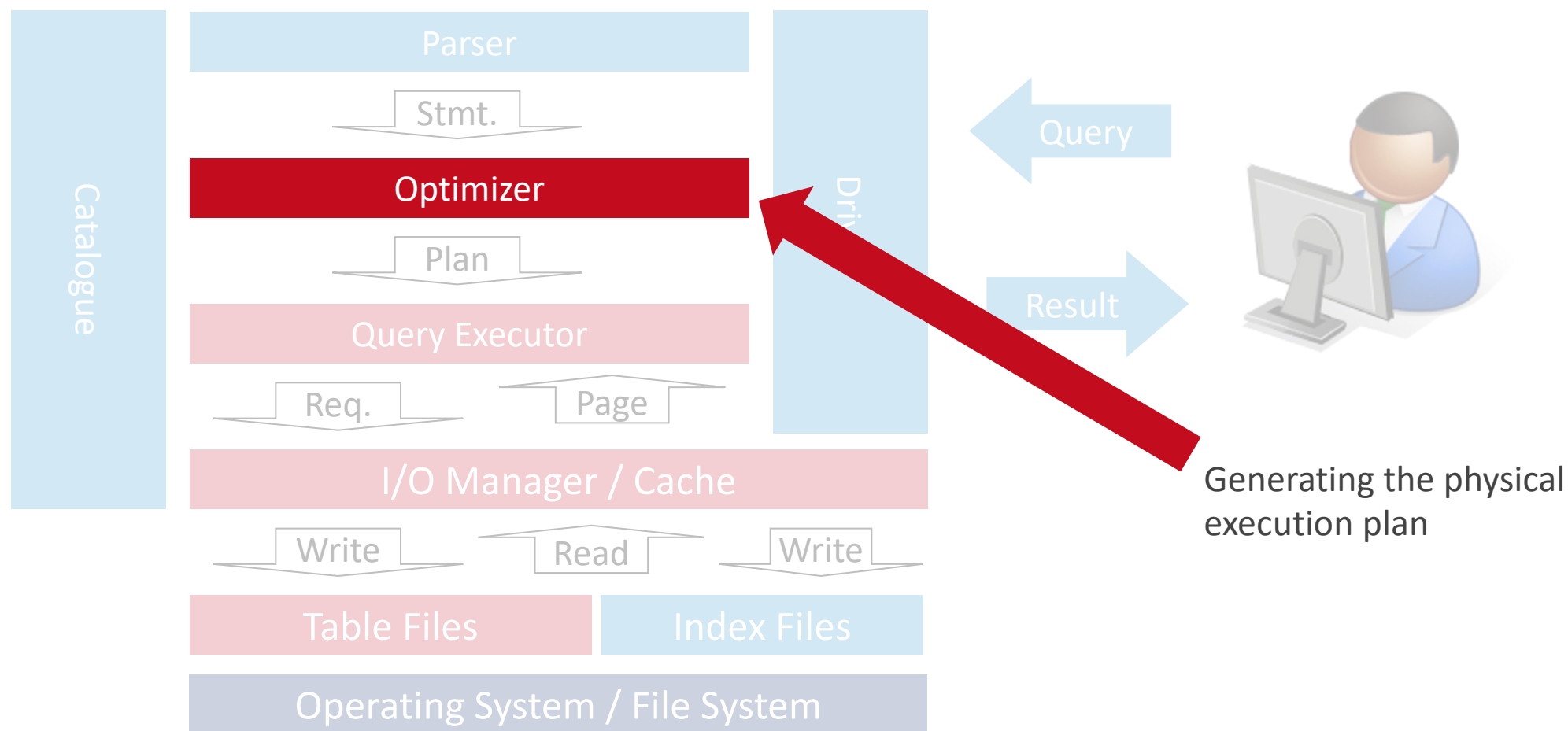
<https://tuport.sap.tu-berlin.de/>

Keep in mind the letter delivery times.

Basic system architecture



Where are we today?



Recap: 4 Steps Approach

- I) Estimation of local cardinalities
 - Cardinalities after single table access and its local predicates
- II) Cardinality-based selection of Join Order
 - Not optimal concerning total costs, but typically quite robust
 - Dynamic Programming: Optimal concerning cost formula (minimizing intermediate results)
- III) Selection of Table Access and Selection of Physical Join Operator
 - Based on I/O cost estimations
- IV) Generation of GroupBy and OrderBy
 - Just added on top of query, no analysis for push-down (sometimes aggregations are possible before the joins)

Recap: Join order optimization

- Decide the order in which tables are joined.
 - $((R \bowtie S) \bowtie T) \bowtie U$ vs. $R \bowtie (S \bowtie (T \bowtie U))$
 - Minimizes (costs of) intermediate results.
- Dynamic programming.
 - Build more complex join plans from intermediate join plans.
 - Use the fastest join plan for each intermediate.
- Result is an abstract join plan.
 - Ignores actual join algorithms.

Plan	Relation / Join Bitmap	Output Cardinality	Operator Costs	Cumulative Costs
S	0001 (1)	200	0	0
PS	0010 (2)	20	0	0
N	0100 (4)	25	0	0
R	1000 (8)	1	0	0
$S \bowtie PS$	0011 (3)	20	220 (200 + 20)	220 (220 + 0 + 0)
$S \bowtie N$	0101 (5)	200	225 (200 + 25)	225 (225 + 0 + 0)
$N \bowtie R$	1100 (12)	5	26 (25 + 1)	26 (26 + 0 + 0)
$(N \bowtie R) \bowtie S$	1101 (13)	40	205 (200 + 5)	231 (205 + 26 + 0)
$(S \bowtie N) \bowtie PS$	0111 (7)	20	220 (200 + 20)	445 (220 + 225 + 0)
$(S \bowtie PS) \bowtie N$	0111 (7)	20	45 (20 + 25)	265 (45 + 220 + 0)
$((N \bowtie R) \bowtie S) \bowtie PS$	1111 (15)	4	60 (40 + 20)	291 (60 + 231 + 0)
$((S \bowtie PS) \bowtie N) \bowtie R$	1111 (15)	4	21 (20 + 1)	286 (21 + 265 + 0)
$(S \bowtie PS) \bowtie (N \bowtie R)$	1111 (15)	4	25 (20 + 5)	271 (25 + 220 + 26)

Recap: Cost estimation

- Determines the costs of an operator
- Costs depend on operator input size and operator properties.
 - Example: Filter selectivity.

What's missing?

1. How to access base tables.
 - Full table scan or use an index?
2. Which join algorithm to use.
 - Nested loop join
 - Index-nested loop join
 - Sort merge join
3. When to sort.
 - Sort early and reuse later?
 - Interacts with the previous two decisions!
4. Inner and outer relation in nested loop joins.

Example: Table and query

– Table Employee

Name	DeptNo	JobNo	Salary
------	--------	-------	--------

- 10000 tuples
- 100 pages
- 500 employees per department

– Table Departments

DeptNo	DeptName	Location
--------	----------	----------

- 20 tuples
- 2 pages
- 3 departments in Denver

– Query

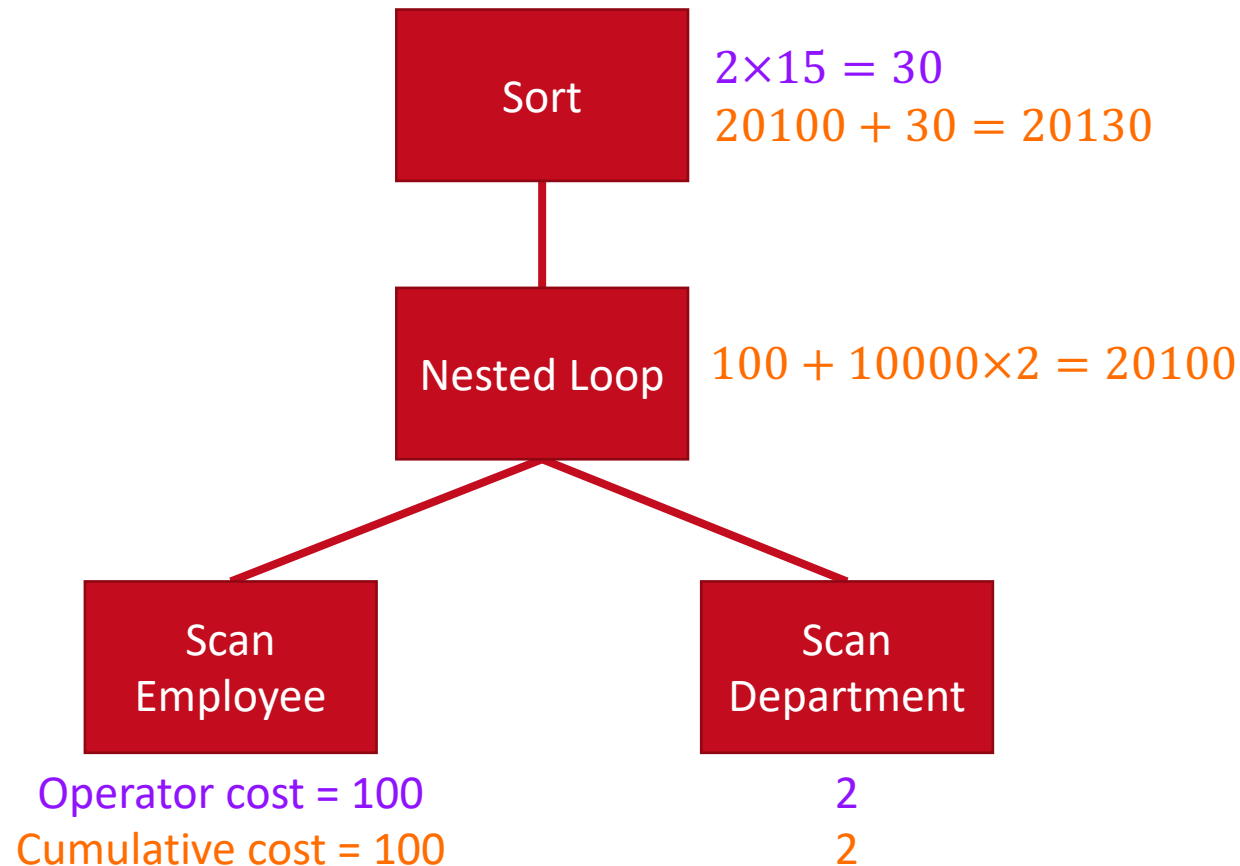
```
SELECT D.DeptName, E.Name
FROM Employee E, Department D
WHERE D.DeptNo = E.DeptNo
AND Location = 'Denver'
ORDER BY D.DeptNo
```

- 1500 result tuples
- 15 pages

Example: Costs

- Assumptions
 - Cost of reading/writing a page is 1
 - No random read/write overhead
- Scan of table with P pages: P
- Sort of table with P pages: $2P + C(child)$
- Nested loop join: $C(outer) + |outer| \times C(inner)$
- Sort merge join: $C(outer) + C(inner)$

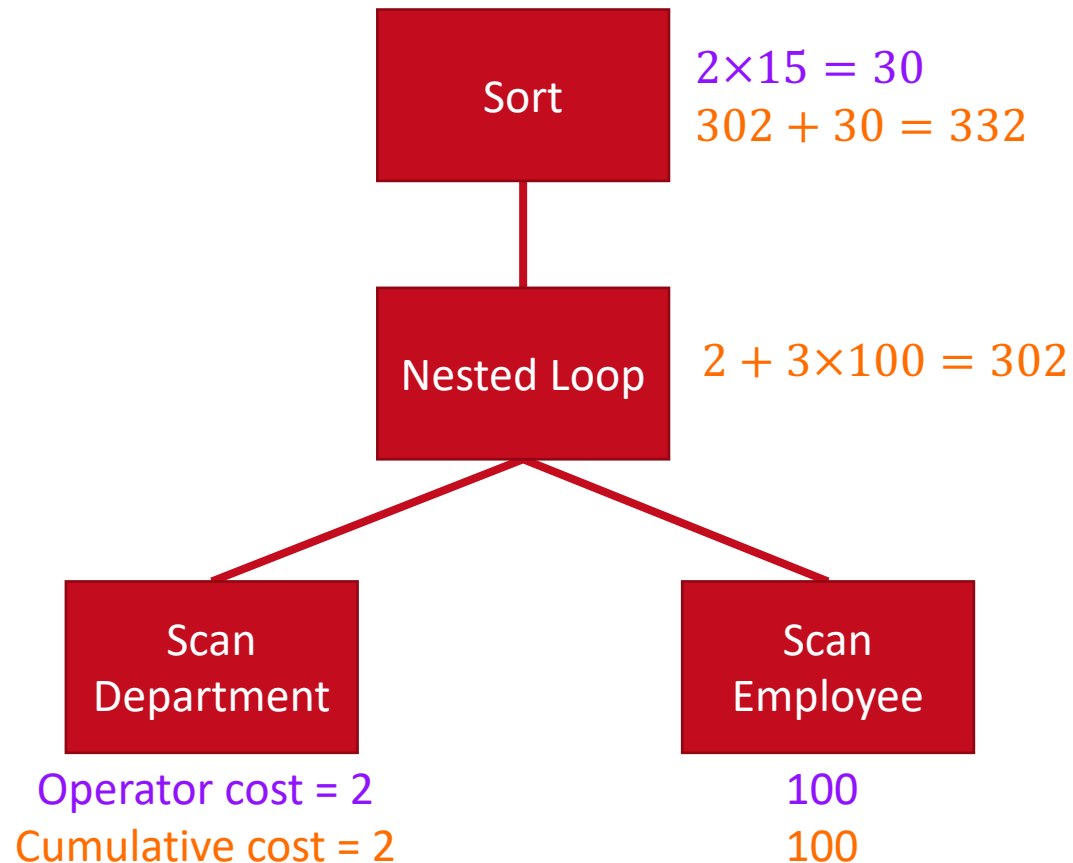
Plan 1: Nested loop with Employee as outer table



- Scan: P
- Sort: $2P + C(child)$
- Nested loop: $C(o) + |o| \times C(i)$
- Sort merge: $C(o) + C(i)$

- $P(Employee) = 100$
- $P(Department) = 2$
- $P(Result) = 15$

Plan 2: Nested loop with Department as outer table

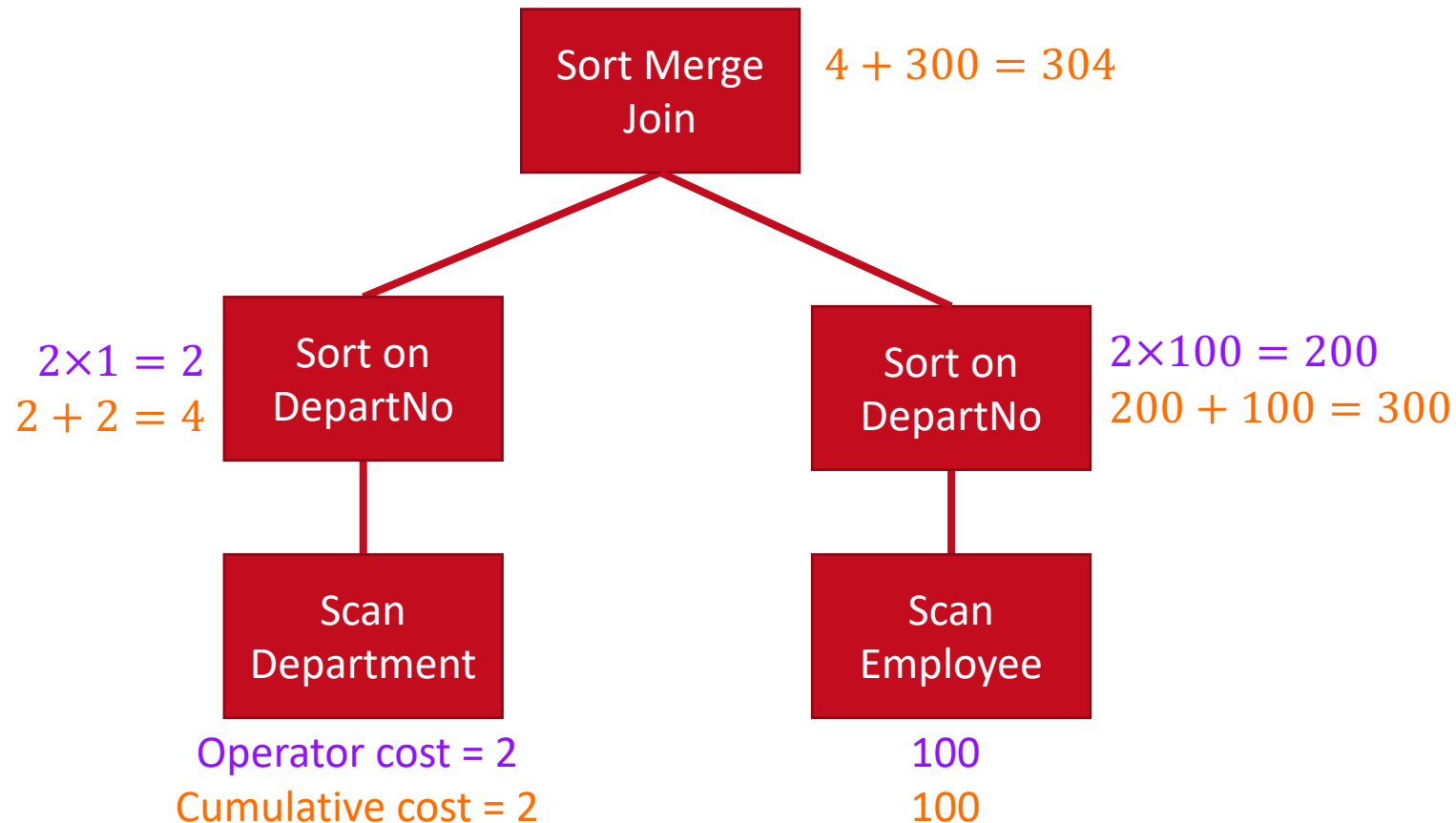


- Scan: P
- Sort: $2P + C(child)$
- Nested loop: $C(o) + |o| \times C(i)$
- Sort merge : $C(o) + C(i)$

- $P(Employee) = 100$
- $P(Department) = 2$
- 3 Departments in Denver
- $P(Result) = 15$

➤ Use smaller table as outer table!

Plan 3: Sort merge join



- Scan: P
- Sort: $2P + C(child)$
- Nested loop: $C(o) + |o| \times C(i)$
- Sort merge: $C(o) + C(i)$

```

SELECT D.DeptName, E.Name
FROM Employee E, Department D
WHERE D.DeptNo = E.DeptNo
AND Location = 'Denver'
ORDER BY DeptNo
  
```

- Sorting earlier can save costs later!
- Sort order for the sort merge join can be reused for the ORDER BY.

When to sort?

- Operations that have to sort their inputs.
 - ORDER BY
 - GROUP BY
 - We use sorted aggregation.
 - Sort merge join
- If the input is already sorted, then these operations are cheaper!
- Operations that produce sorted output.
 - SORT (obviously)
 - GROUP BY (sorted on grouping columns)
 - Sort merge join (sorted on join columns)
 - Index lookup (sorted on the key columns)
- Operations later in the plan may benefit from a more expensive operation earlier in the plan.

Interesting orders

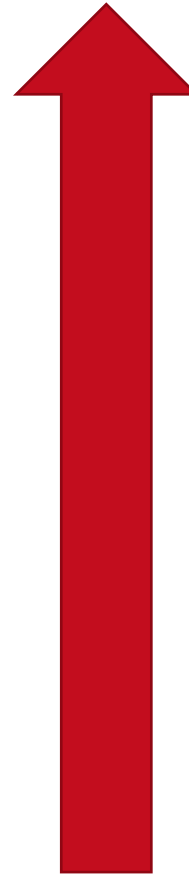
- Communicate the intent to create a useful sort order which can be used later in the plan.
 - On which columns to sort.
 - Cost delta: How much it would cost to sort the unsorted input.
 - Depends on the input cardinality.
 - Pass interesting orders down to the subplan(s) if the subplans produce the columns that should be sorted.
- Example: Join $R.a = S.b$
 - R has 10 pages $\rightarrow$ Interesting order on $R.a$, cost delta = $2 \times 10 = 20$
 - S has 20 pages $\rightarrow$ Interesting order on $S.b$, cost delta = $2 \times 20 = 40$

Top-down / bottom-up plan generation



Phase 1: Top-down

1. Start at the top and recursively descend the operator tree.
2. For each operator that requires sorting.
 - Create interesting order.
 - Pass applicable interesting orders to subplans.



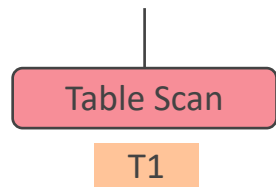
Phase 2: Bottom-up

3. For each base relation, generate access plans.
 - Cheapest plan.
 - Cheapest plan that satisfies an interesting order.
 - Produces the tuples in the required order.
 - Additional costs are less than the cost delta.
4. For each join, join child access plans with different join algorithms and prune.
 - Retain cheapest plan.
 - Retain cheapest plans which satisfy an interesting order.

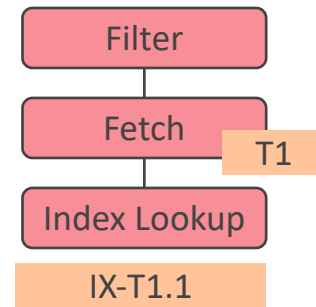
Generating base table access plans

- Input:
 - BaseTableAccess instance.
 - Contains cardinalities but does not yet contain costs.
 - Zero or more interesting orders with cost delta.
- Output: Set of plans with associated costs
 - Cheapest plan
 - For each interesting order: Cheapest plan which satisfies it.
 - Creates tuples in the required order.
 - Additional costs (compared to cheapest plan) are smaller than the cost delta.
- Possible plans depend on local predicates and available indexes.
 - In general, an index can be used for one or more predicates when the columns referenced in the predicates are an initial substring of the index key columns.
 - Our system only supports single-key indexes.
 - Available indexes can be retrieved from the catalog.

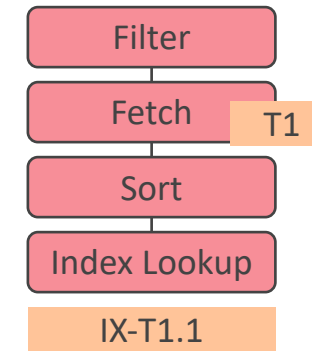
Possible base relation access plans



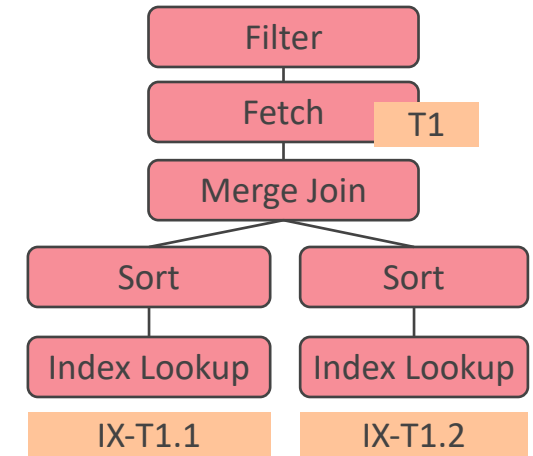
- Full sequential scan.
- Evaluate local predicates.
- Produces an unordered plan (sorted on RID).



- Evaluate predicates which are an initial substring of the index key columns.
- **Random access** to tuples in T1 during Fetch.
- **Results sorted** on key columns.



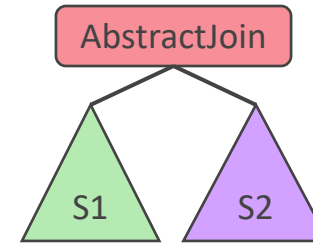
- **Sequential access** to tuples in T1 during Fetch.
- **Results not sorted** on key columns.



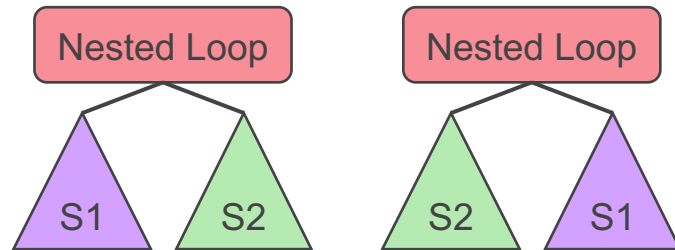
- Use multiple indexes to evaluate predicates.
- Results not sorted on key columns.
- Not used in exercises.

Generating join plans

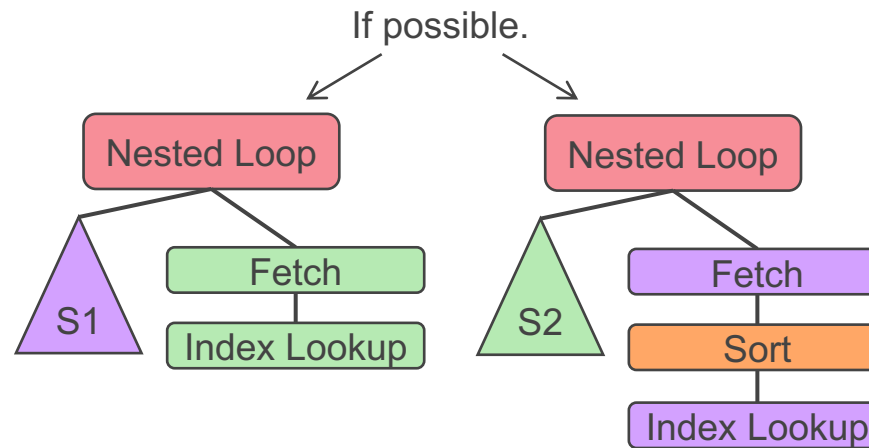
- Input
 - AbstractJoinOperator instance
 - Does not yet contain costs.
 - Does not specify inner and outer relations.
 - Set of interesting orders
 - Candidate plans for both children
- Output: Set of plans with associated costs
 - Cheapest plan
 - For each interesting order: Cheapest plan which satisfies it.



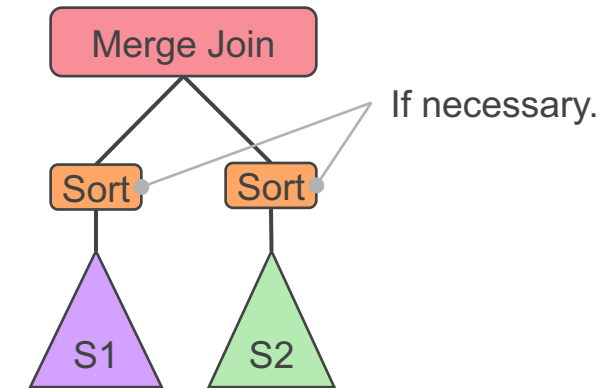
Possible join plans



- Works for any combination of child subplans and join predicates.
- Generally only return only one.
 - Either one is cheaper or both cost the same.
- However: The more expensive plan could satisfy an interesting order!



- If one of the subplans is a base table.
- Return only if cheaper than table scan.
- Sort RIDs before Fetch if that reduces costs.



- Works for any combination of child subplans.
- Works best for equi joins.
- Only sort subplans if they are not already sorted.

PhysicalPlanGenerator API

PhysicalPlanGenerator interface

- Contains six methods that together implement step III and step IV of the optimization process.

- `OptimizerPlanOperator generatePhysicalPlan(...)`
- `OptimizerPlanOperator[] buildBestOrderByPlans(...)`
- `OptimizerPlanOperator[] buildBestGroupByPlans(...)`
- `OptimizerPlanOperator[] buildBestSubPlans(...)`
- `OptimizerPlanOperator[] buildBestConcreteJoinPlans(...)`
- `OptimizerPlanOperator[] buildBestRelationAccessPlans(...)`

Calls

Calls

Calls

Calls for
subplans

Calls either
method depending
on subplan type

PhysicalPlanGenerator interface: Create final physical plan

```
OptimizerPlanOperator generatePhysicalPlan(AnalyzedSelectQuery query,  
                                           OptimizerPlanOperator joinPlan);
```

- Public method called by the optimizer to create a concrete physical execution plan.
- `query`: contains information about the produced columns; if there are ORDER BY, GROUP BY, or HAVING clauses; etc.
- `joinPlan`: contains the join order that was determined earlier in step II.
- Calls `buildBestOrderByPlans` to create candidate plans.

PhysicalPlanGenerator interface: ORDER BY

```
OptimizerPlanOperator[] buildBestOrderByPlans(AnalyzedSelectQuery query,  
                                              OptimizerPlanOperator joinPlan);
```

- Create candidate plans that satisfy the ORDER BY clause if present.
 - Sort order and direction is important!
- Call `buildBestGroupByPlans` to process subplans.
- Reuse sort for a GROUP BY if possible (see next slide).

PhysicalPlanGenerator interface: GROUP BY and HAVING

```
OptimizerPlanOperator[] buildBestGroupByPlans(ProducedColumn[] outCols,  
                                              boolean grouping,  
                                              LocalPredicate havingPred,  
                                              OptimizerPlanOperator joinPlan,  
                                              RequestedOrder[] order,  
                                              int[] orderColIndices);
```

- Create candidate plans that satisfy the GROUP BY and HAVING clauses if present.
- `outCols`: Columns produced by the query (SELECT clause).
- `grouping`: Indicate if there is a GROUP BY clause.
- `havingPred`: Indicate if there is a HAVING clause.
- `order`: Order imposed by the ORDER BY clause if the sorted columns are a subset of the grouping columns.
 - Otherwise cannot reuse ORDER BY because sort aggregated columns are sorted before grouping columns.
- `orderColIndices`: The position of the columns referenced by the ORDER BY clause in the final SELECT clause.
- Call `buildBestSubPlans` to process subplans.

PhysicalPlanGenerator interface: Process subplans

```
OptimizerPlanOperator[] buildBestSubPlans(Column[] neededCols,  
                                           OptimizerPlanOperator abstractJoinPlan,  
                                           InterestingOrder[] intOrders);
```

- Process either a base table or an (intermediate) join.
 - Call `buildBestConcreteJoinPlans` or `buildBestRelationAccessPlans`.
- `neededCols`: Columns that this subplan needs to return.
 - Columns used in SELECT, WHERE, ORDER BY, or GROUP BY clauses.
 - Columns used in joins.
- `intOrders`: Interesting orders that this subplan should satisfy.

PhysicalPlanGenerator interface: Process joins

```
OptimizerPlanOperator[] buildBestConcreteJoinPlans(  
    Column[] neededCols,  
    AbstractJoinPlanOperator abstractJoinPlan,  
    InterestingOrder[] intOrders);
```

- Create join plan candidates according to slide 18.
 - Merge join if it is an equi join.
 - Nested loop joins with both orders if merge join is not possible.
 - Indexed nested loop join if one child is a base table.
- Retain only the cheapest plan and any cheapest plan that satisfies an interesting order.
- `neededCols`: Columns required from both join sides.
 - Split array to columns to corresponding subplan.
- Create interesting orders for subplans.
 - Any sort order and direction works.
- Call `buildBestSubPlans` to process subplans.

PhysicalPlanGenerator interface: Process base tables

```
OptimizerPlanOperator[] buildBestRelationAccessPlans(Column[] neededCols,  
                                                    Relation abstractJoinPlan,  
                                                    InterestingOrder[] intOrders);
```

- Create base table access candidates according to slide 16.
 - Table Scan
 - Index Lookup + Fetch
 - Index Lookup + Sort + Fetch
- Retain only the cheapest plan and any cheapest plan that satisfies an interesting order.

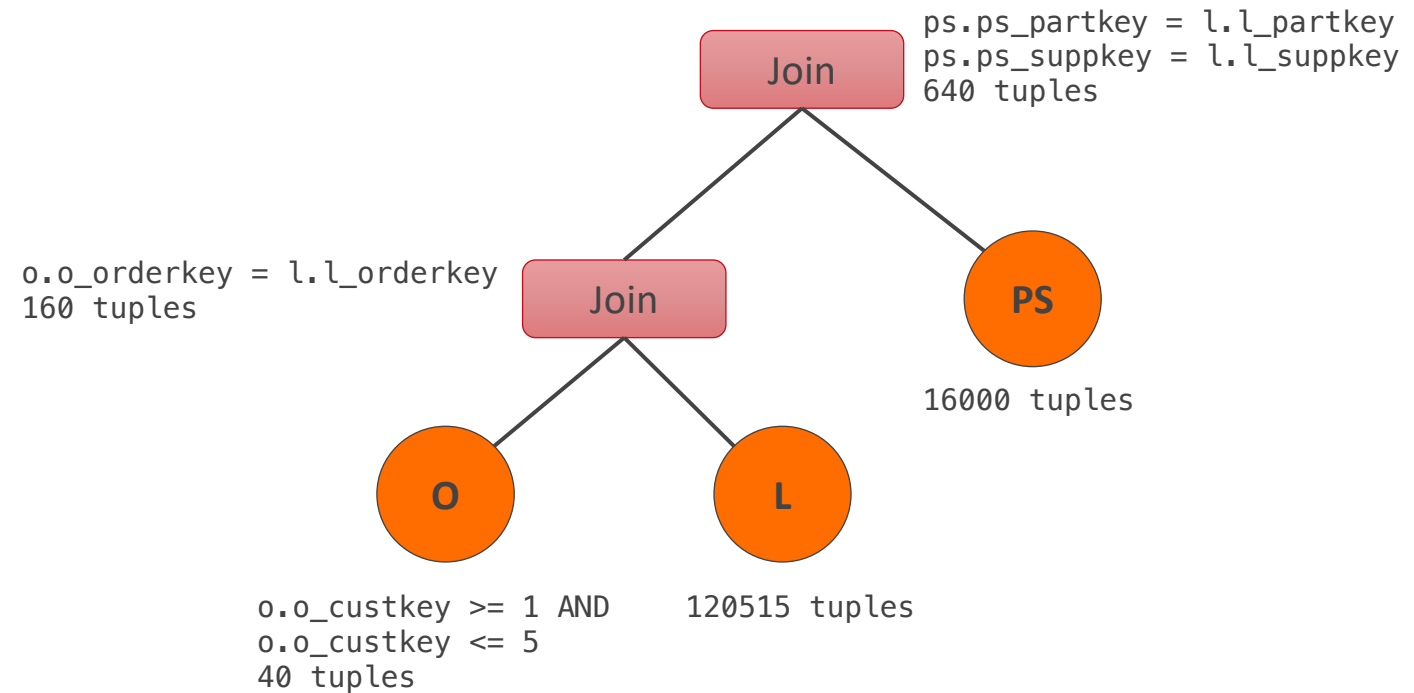
Examples

Example #1

– Query

```
SELECT
    ps.ps_availqty as ps_availqty,
    l.l_quantity   as l_quantity,
    o.o_orderdate  as o_orderdate
FROM
    partsupplier ps,
    lineitem l,
    orders o
WHERE
    ps.ps_partkey = l.l_partkey AND
    ps.ps_suppkey = l.l_suppkey AND
    o.o_orderkey = l.l_orderkey AND
    o.o_custkey >= 1 AND
    o.o_custkey <= 5;
```

– Optimized join plan and estimated cardinalities



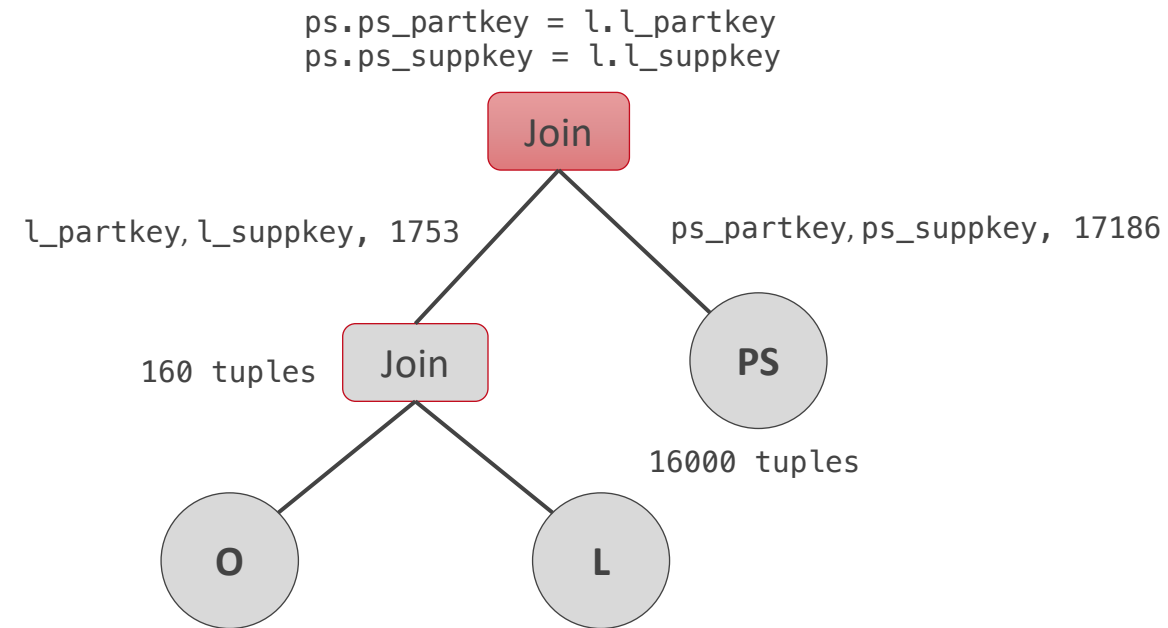
Example #1: ORDER BY and GROUP BY

- No ORDER BY and GROUP BY clause
- `buildBestOrderByPlans` and `buildBestGroupByPlans` just pass along results

```
SELECT
    ps.ps_availqty as ps_availqty,
    l.l_quantity   as l_quantity,
    o.o_orderdate  as o_orderdate
FROM
    partsupplier ps,
    lineitem l,
    orders o
WHERE
    ps.ps_partkey = l.l_partkey AND
    ps.ps_suppkey = l.l_suppkey AND
    o.o_orderkey = l.l_orderkey AND
    o.o_custkey >= 1 AND
    o.o_custkey <= 5;
```

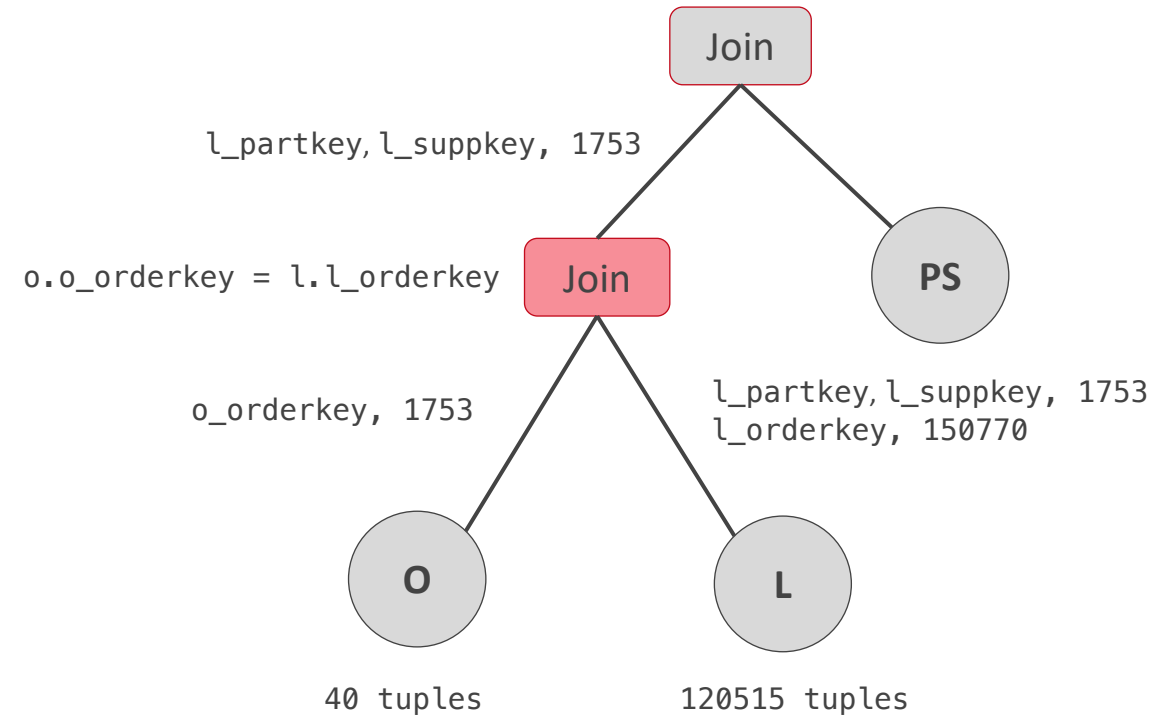
Example #1: Interesting orders for first join

- Requires two columns from each side
 - Left side: `l_partkey`, `l_suppkey`
 - Right side: `ps_partkey`, `ps_suppkey`
- Cost to sort 160 tuples from left side: 1753
- Cost to sort 16000 tuples from right side: 17186
- Interesting order passed to left side:
`l_partkey, l_suppkey, 1753`
- Interesting order passed to right side:
`ps_partkey, ps_suppkey, 17186`



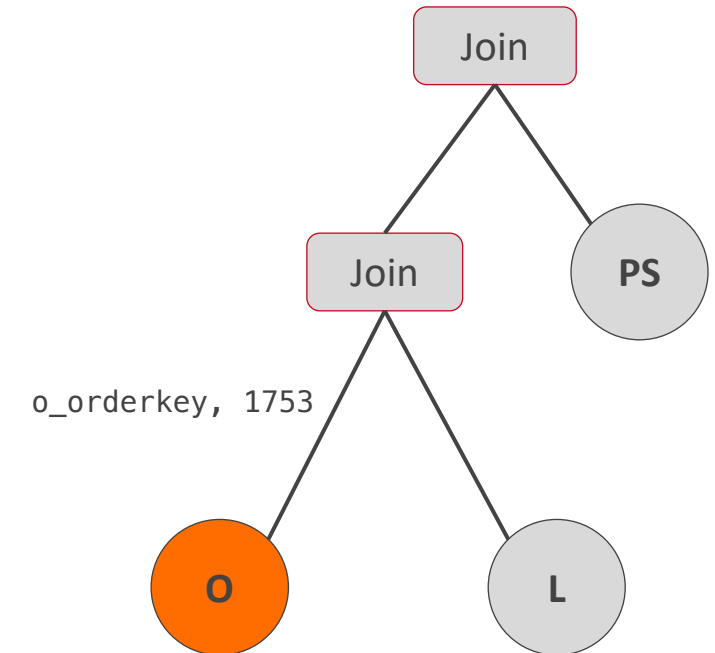
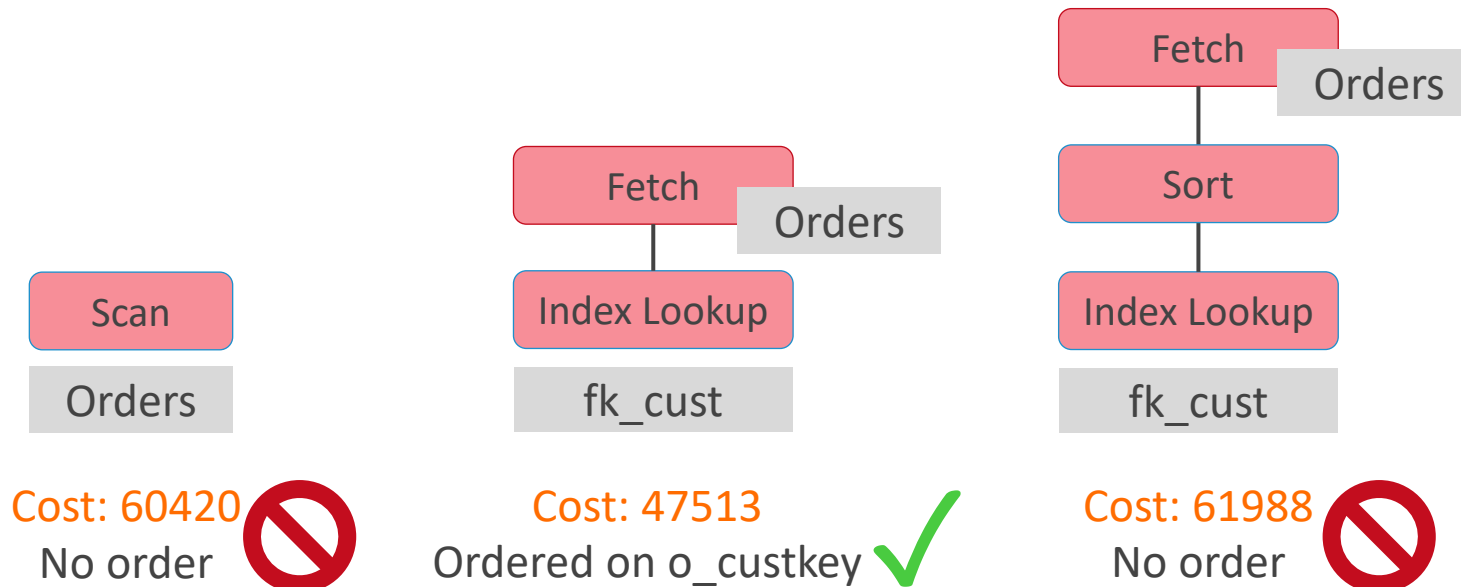
Example #1: Interesting orders for second join

- Requires one column from each side
 - Left side: o_orderkey
 - Right side: l_orderkey
- Cost to sort 40 tuples from left side: 1753
- Cost to sort 120515 tuples from right side: 150770
- Interesting order from previous join
l_partkey, l_suppkey, 1753
- Interesting order passed to left side:
o_orderkey, 1753
- Interesting order passed to right side:
l_partkey, l_suppkey, 1753
l_orderkey, 150770



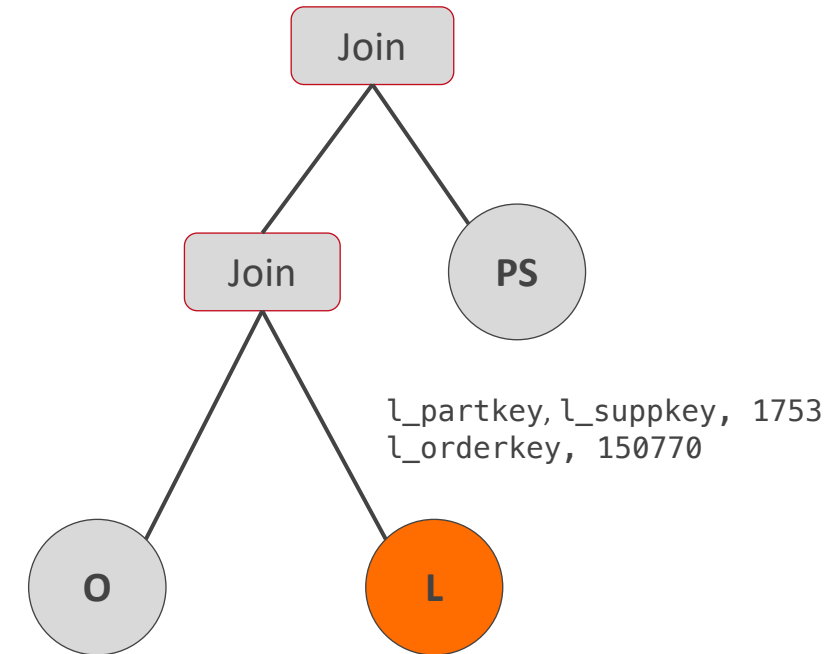
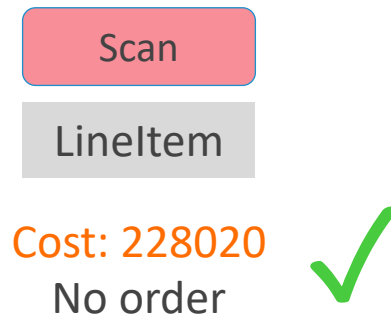
Example #1: Base table access on Orders

- Query contains predicate on o_custkey
- Three plans are possible.
- No plan satisfies the interesting order o_orderkey, 1753
- Index Lookup + Fetch is the cheapest plan and is retained.



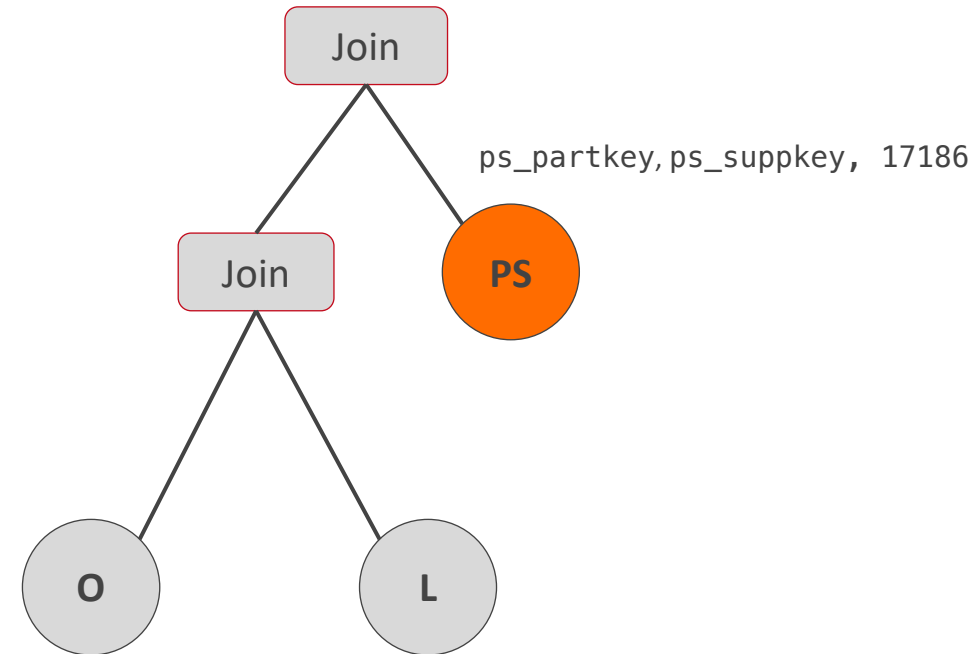
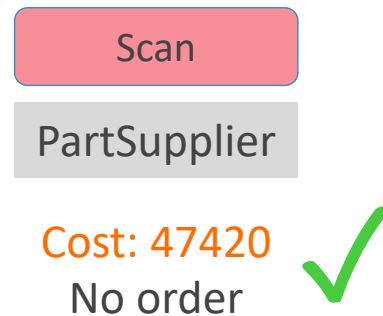
Example #1: Base table access on Lineltem

- No predicate.
- Only scan plan is possible and retained.
- Scan does satisfy any of the interesting orders.



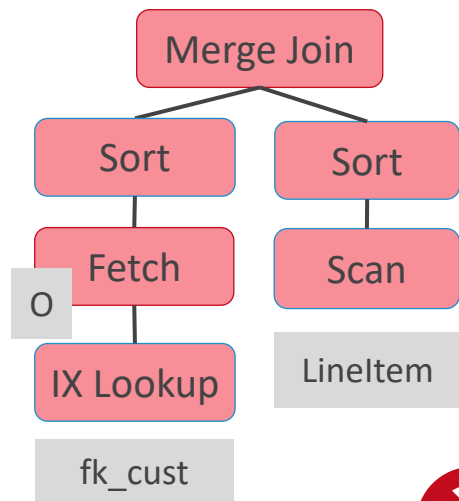
Example #1: Base table access on PartSupplier

- No predicate.
- Only scan plan is possible and retained.
- Scan does satisfy any of the interesting orders.

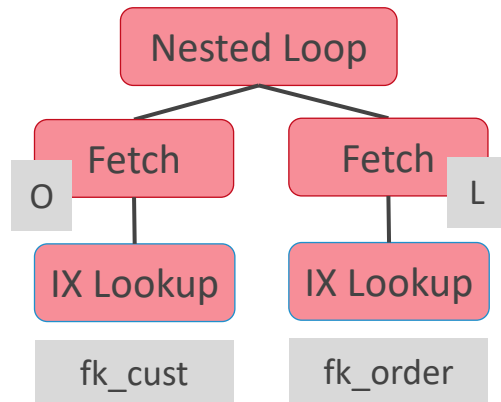


Example #1: Inner join

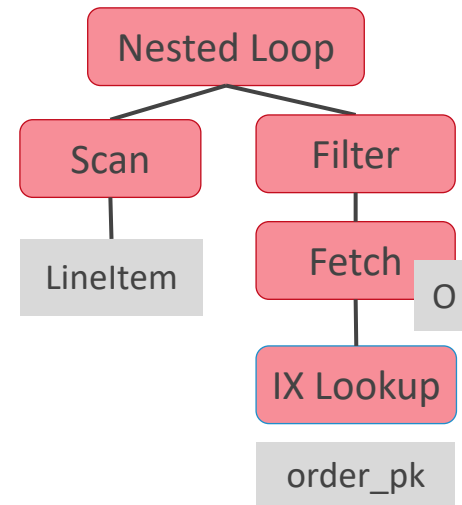
- Have to sort both inputs for merge join.
- Have to use a filter for the predicate on o_custkey for the second index nested loop join.



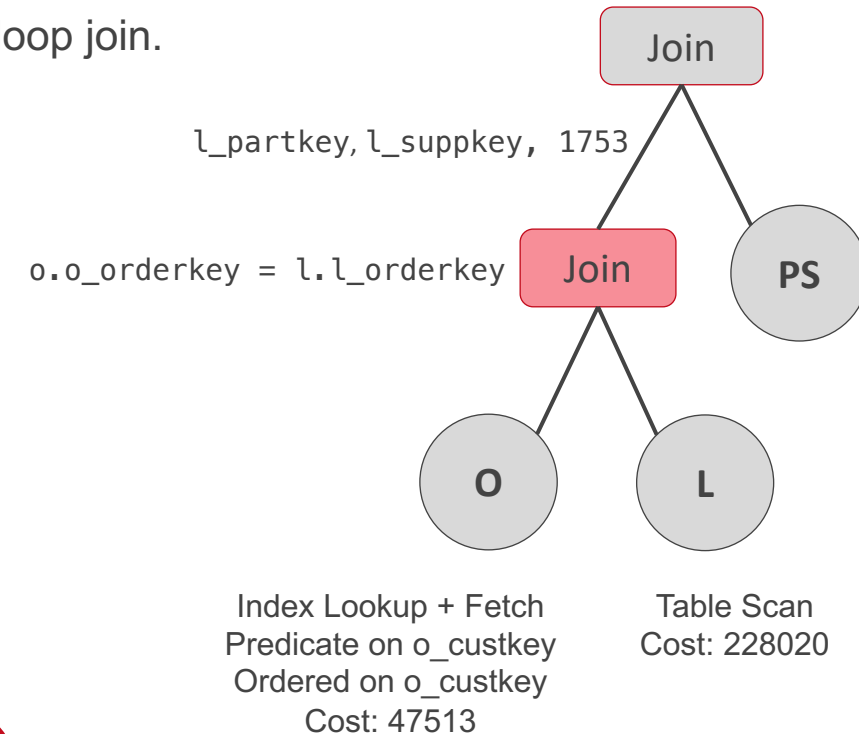
Cost: 428056
Ordered on o_orderkey



Cost: 396513
Ordered on o_custkey

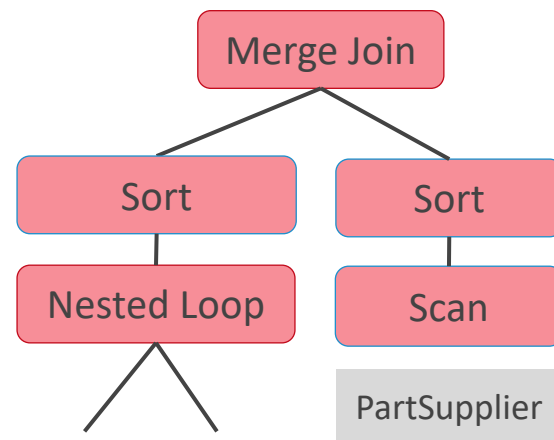


Cost: 371414220
No order



Example #1: Outer join

- Have to sort both inputs for merge join.
- Index nested loop join would theoretically possible but a helper method does not currently support them with conjunct predicates.

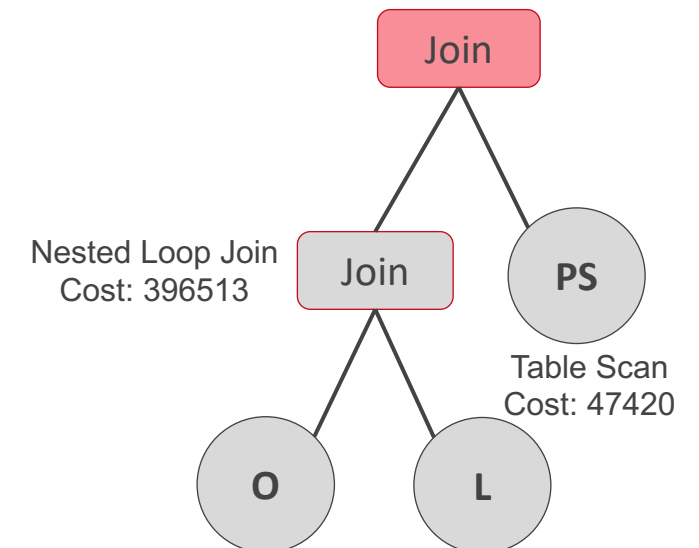


Cost: 462872

Ordered on ps_partkey, ps_suppkey



ps.ps_partkey = l.l_partkey
ps.ps_suppkey = l.l_suppkey



Example #2

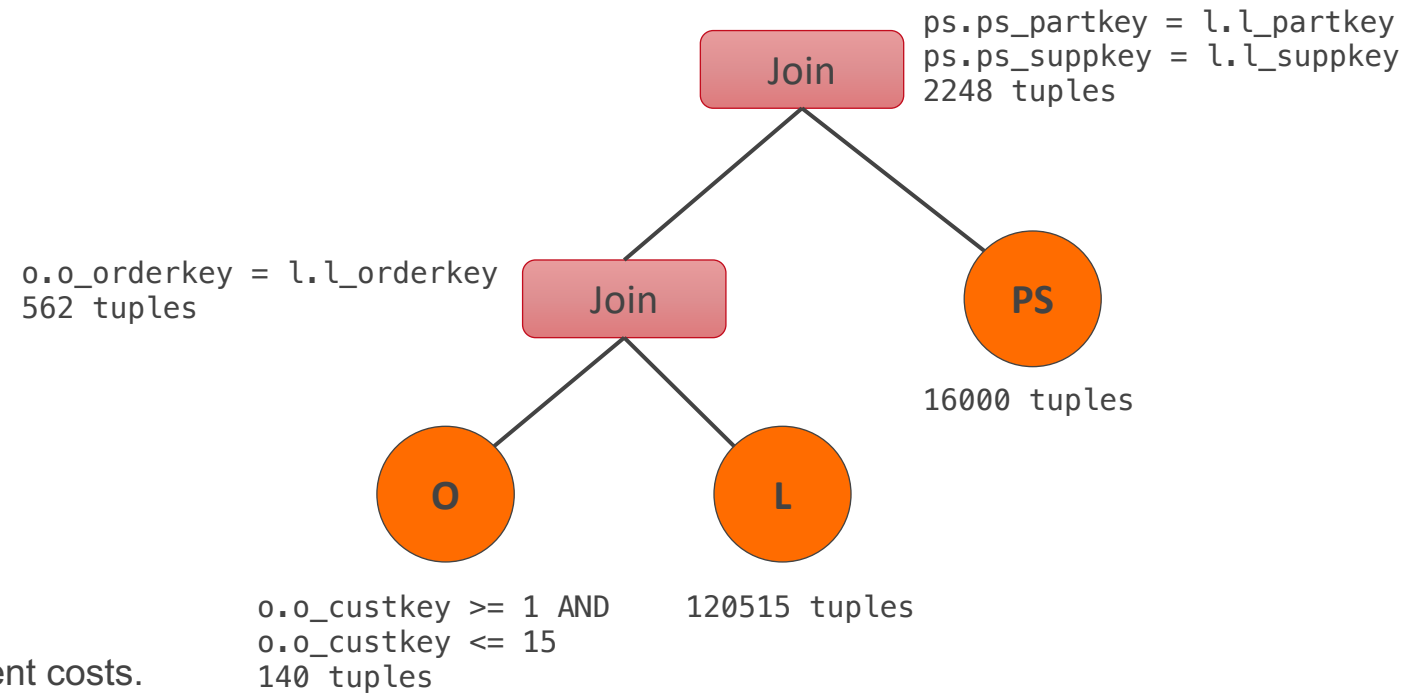
- Query

```

SELECT
    ps.ps_availqty as ps_availqty,
    l.l_quantity   as l_quantity,
    o.o_orderdate  as o_orderdate
FROM
    partsupplier ps,
    lineitem l,
    orders o
WHERE
    ps.ps_partkey = l.l_partkey AND
    ps.ps_suppkey = l.l_suppkey AND
    o.o_orderkey = l.l_orderkey AND
    o.o_custkey >= 1 AND
    o.o_custkey <= 15;
  
```

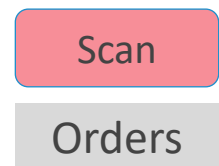
- Less selective local predicate on Orders.
- Interesting orders on the same columns but maybe different costs.
- Base table access for Lineitem and PartSupplier are the same.

- Same join plan but different estimated cardinalities

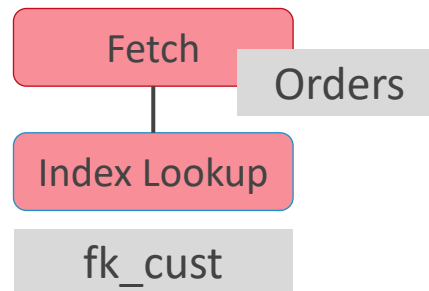


Example #2: Base table access on Orders

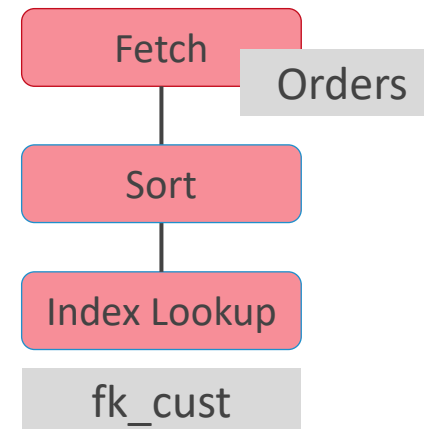
- Predicate is less selective.
- Table scan is now the cheapest plan.



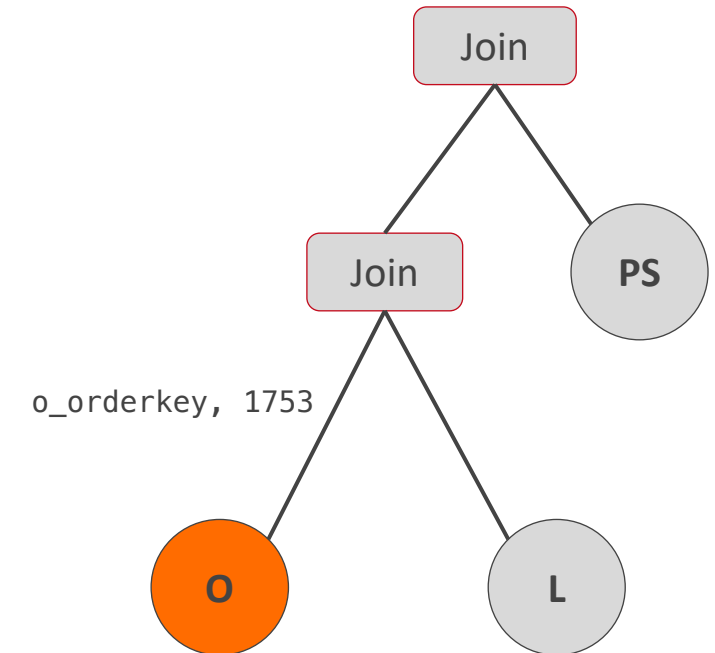
Cost: 60420
No order



Cost: 143874
Ordered on o_custkey

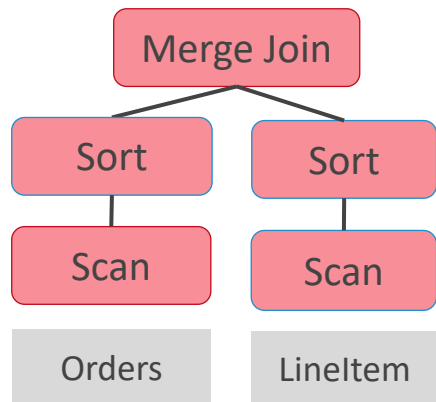


Cost: 192286
No order

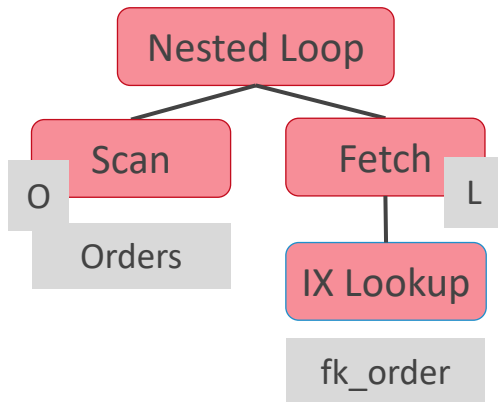


Example #2: Inner join

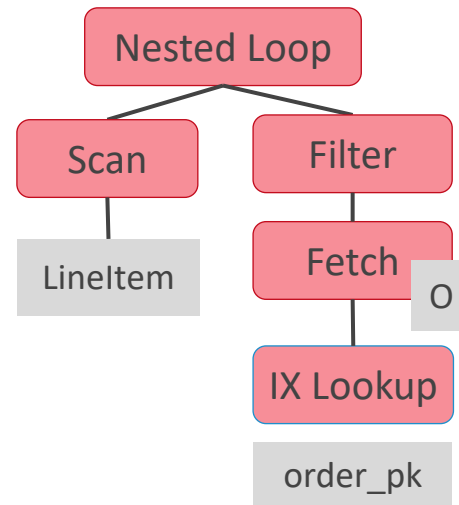
- Subplan for Orders is now a table scan.
- Third plan is unchanged (including costs).
- Merge join is now the cheapest plan.



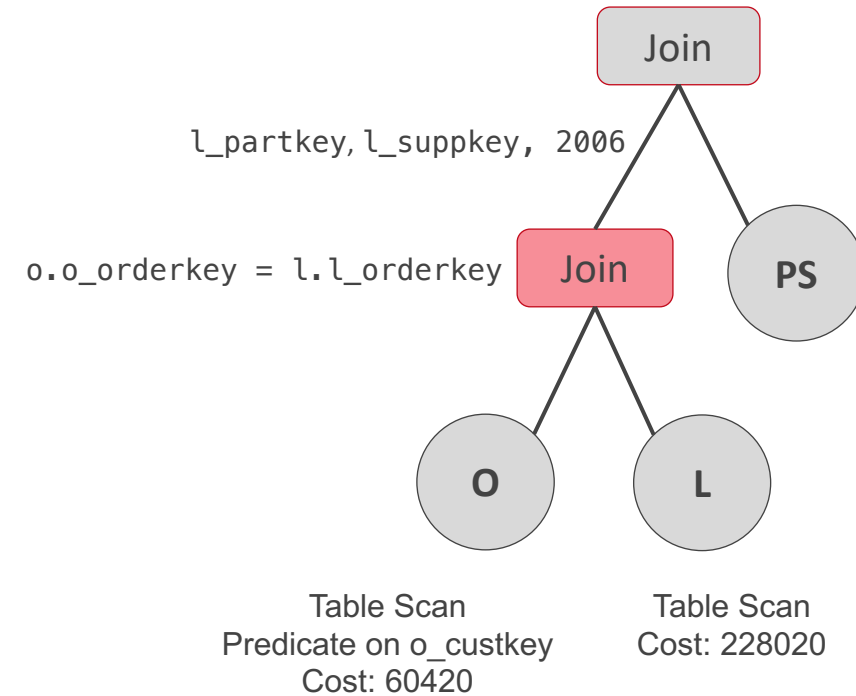
Cost: 440963
Ordered on o_orderkey ✓



Cost: 1281920
No order



Cost: 371414220
No order



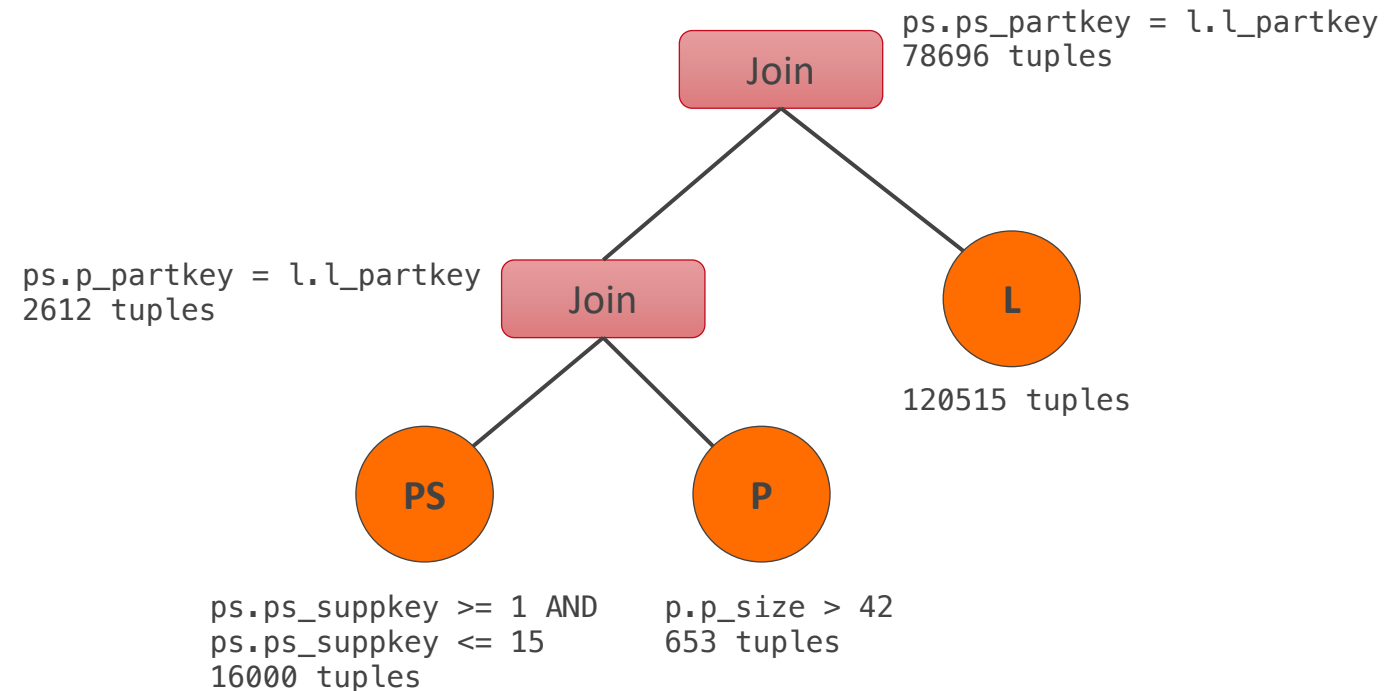
Example #3

- Query

```
SELECT
  p.p_partkey as p_partkey,
  p.p_name as p_name,
  ps.ps_availqty as ps_availqty,
  l.l_quantity as l_quantity,
  l.l_tax as l_tax
FROM
  partsupplier ps, part p, lineitem l
WHERE
  p.p_partkey = ps.ps_partkey AND
  ps.ps_partkey = l.l_partkey AND
  ps.ps_suppkey >= 1 AND
  ps.ps_suppkey <= 200 AND
  p.p_size > 42
ORDER BY
  p_partkey ASC;
```

- ORDER BY clause on attribute that is also used in join.

- Optimized join plan and estimated cardinalities



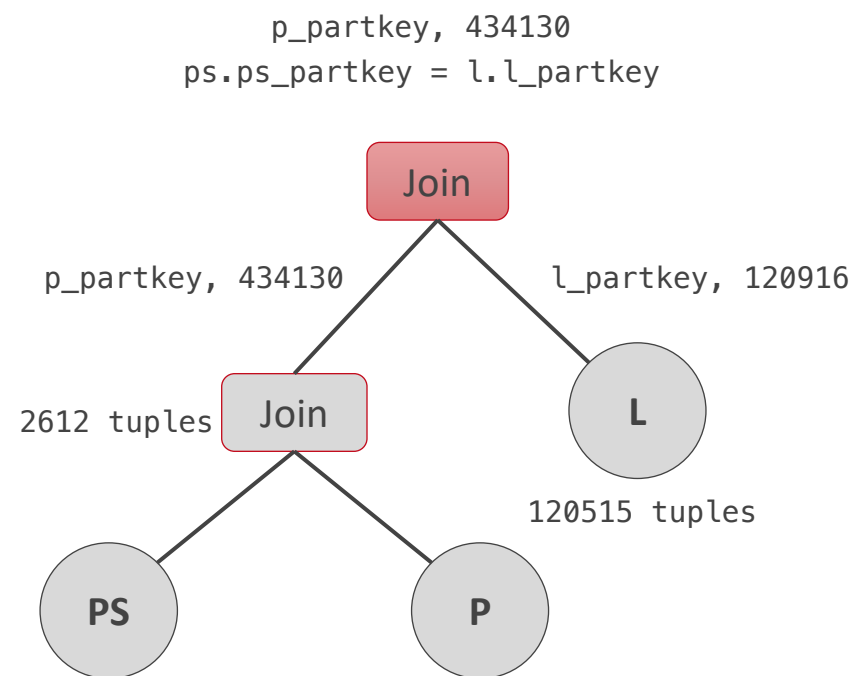
Example #3: ORDER BY clause

- ORDER BY clause on `p.p_partkey`.
- Cost to sort 78696 tuples in final result: 434130.
- Interesting order: `p.p_partkey`, 434130

```
SELECT
    p.p_partkey as p_partkey,
    p.p_name as p_name,
    ps.ps_availqty as ps_availqty,
    l.l_quantity as l_quantity,
    l.l_tax as l_tax
FROM
    partsupplier ps, part p, lineitem l
WHERE
    p.p_partkey = ps.ps_partkey AND
    ps.ps_partkey = l.l_partkey AND
    ps.ps_suppkey >= 1 AND
    ps.ps_suppkey <= 200 AND
    p.p_size > 42
ORDER BY
    p.p_partkey ASC;
```

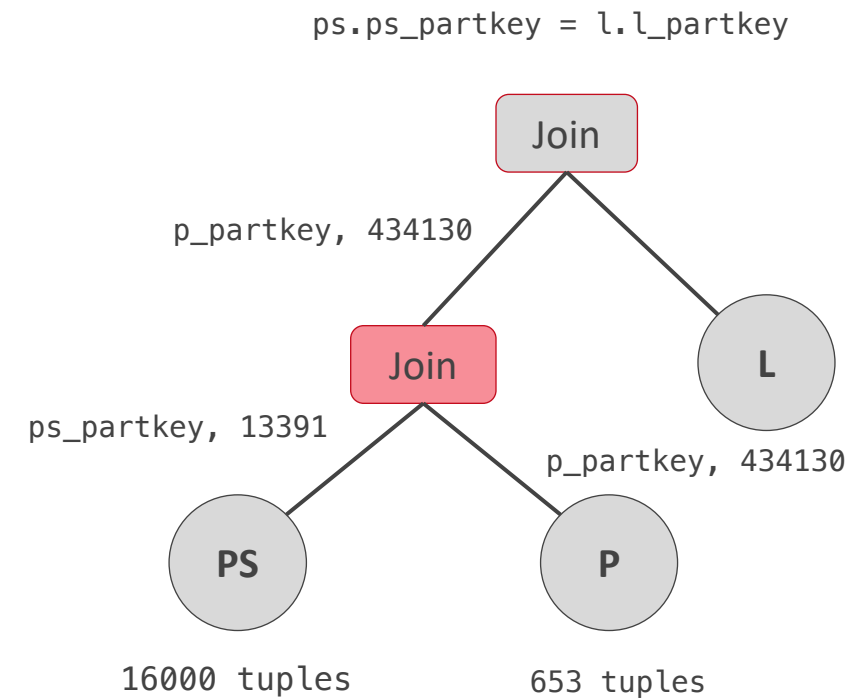
Example #3: Interesting orders for first join

- Interesting order from ORDER BY: $p.p_partkey, 434130$
- Cost to sort 2612 tuples from left side: 14403
 - Table part is contained in the left child.
 - Use existing sorting costs on the same column.
- Cost to sort 120515 tuples from right side: 120916
- Interesting order passed to left side: $p_partkey, 434130$
- Interesting order passed to right side: $ps_partkey, 120916$



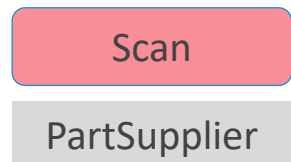
Example #3: Interesting orders for second join

- Interesting order from previous join: $p.p_partkey, 434130$
- Cost to sort 16000 tuples from left side: 13391
- Cost to sort 653 tuples from right side: 4536
 - Use existing sorting costs on the same column.
- Interesting order passed to left side: $ps_partkey, 13391$
- Interesting order passed to right side: $p_partkey, 434130$

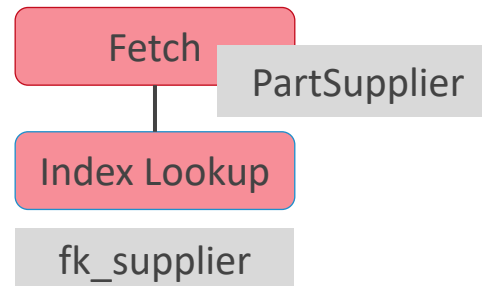


Example #3: Base table access on PartSupplier

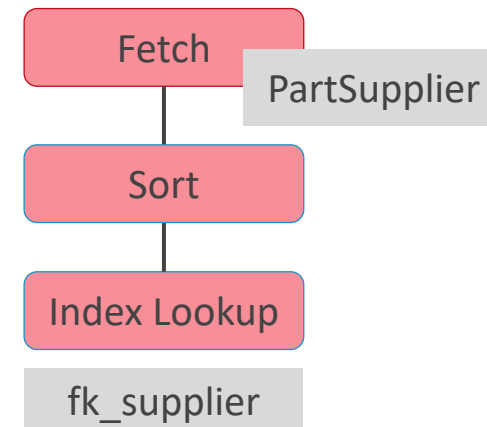
- Query contains predicate on ps_suppkey
- Scan is the cheapest plan and is retained.
- Index Lookup + Sort is faster than Index Lookup.



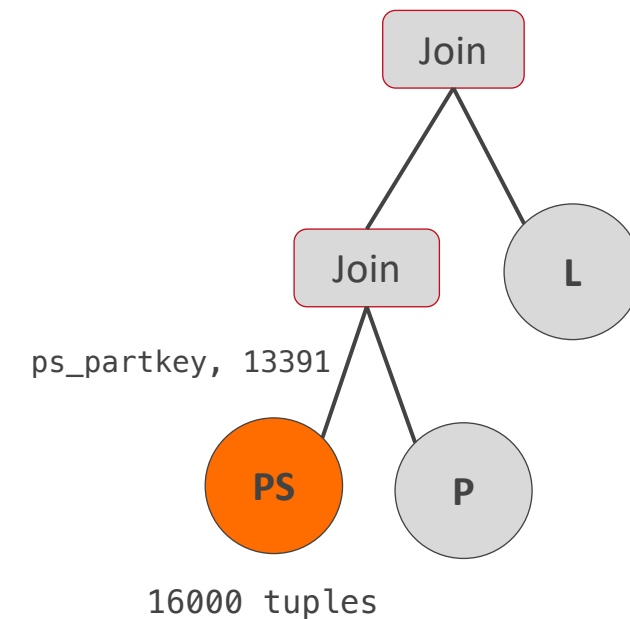
Cost: 47420
No order ✓



Cost: 7804263
Ordered on ps_suppkey ✗

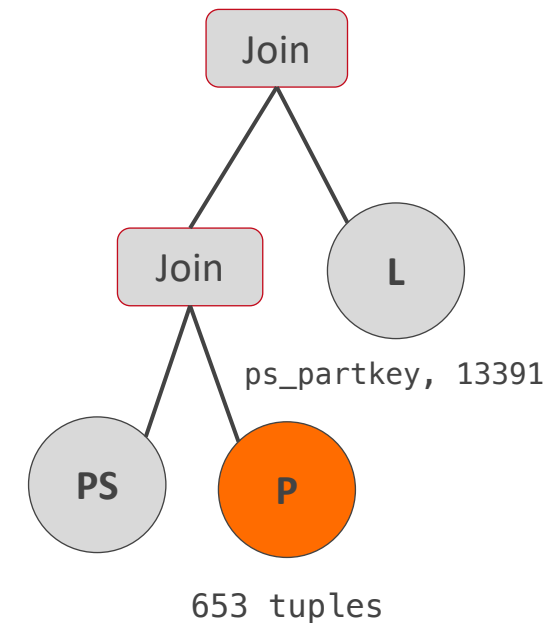
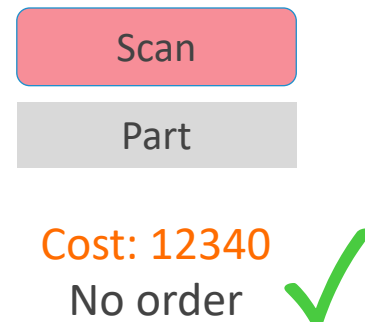


Cost: 277063
No order ✗



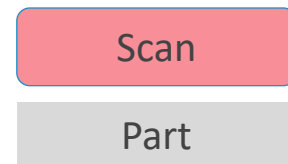
Example #3: Base table access on Parts

- Query contains predicate `p_part < 42` but it's an unbounded range predicate.
- Scan is the only plan.

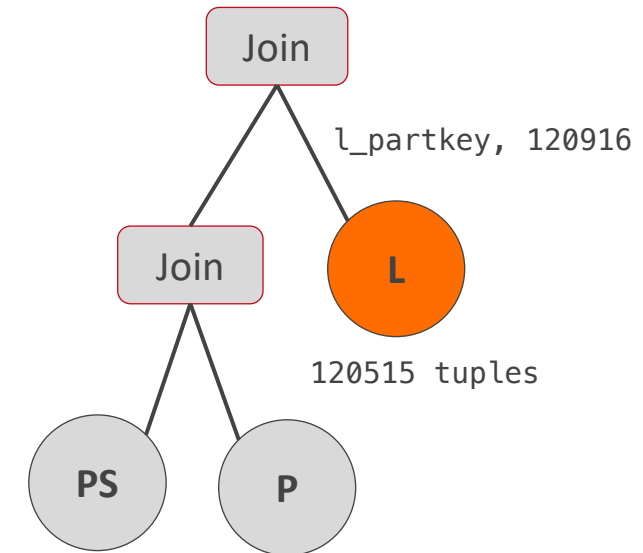


Example #3: Base table access on LineItem

- No predicate.
- Scan is the only plan.

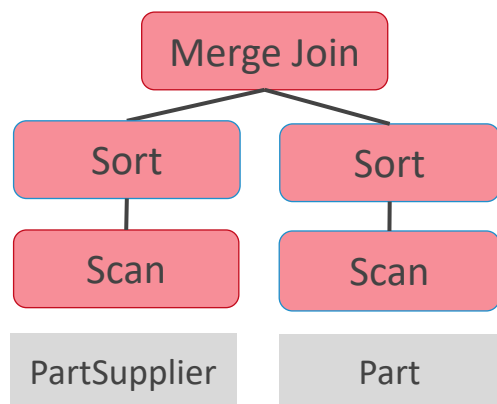


Cost: 228020
No order ✓

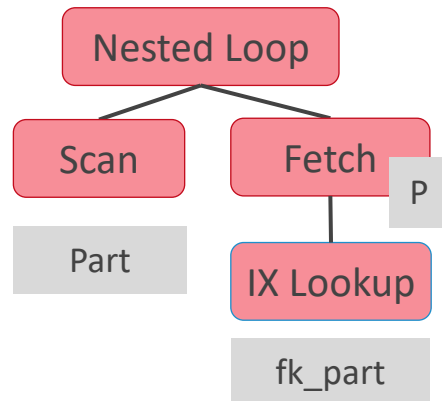


Example #3: Inner join

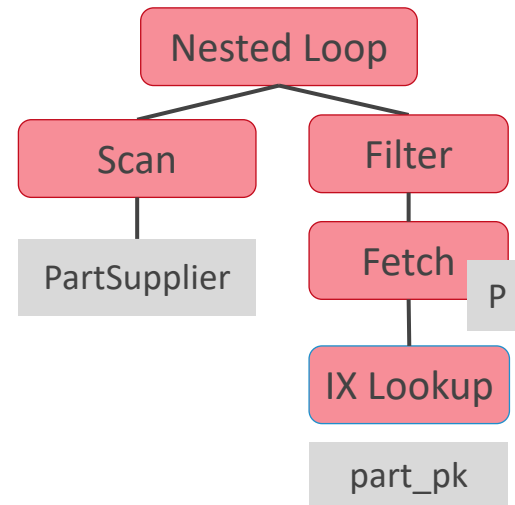
- Have to sort both inputs for merge join.
- Result on merge join satisfies interesting order.
- Have to use a filter for the predicate on ps_custkey for the index nested loop join.



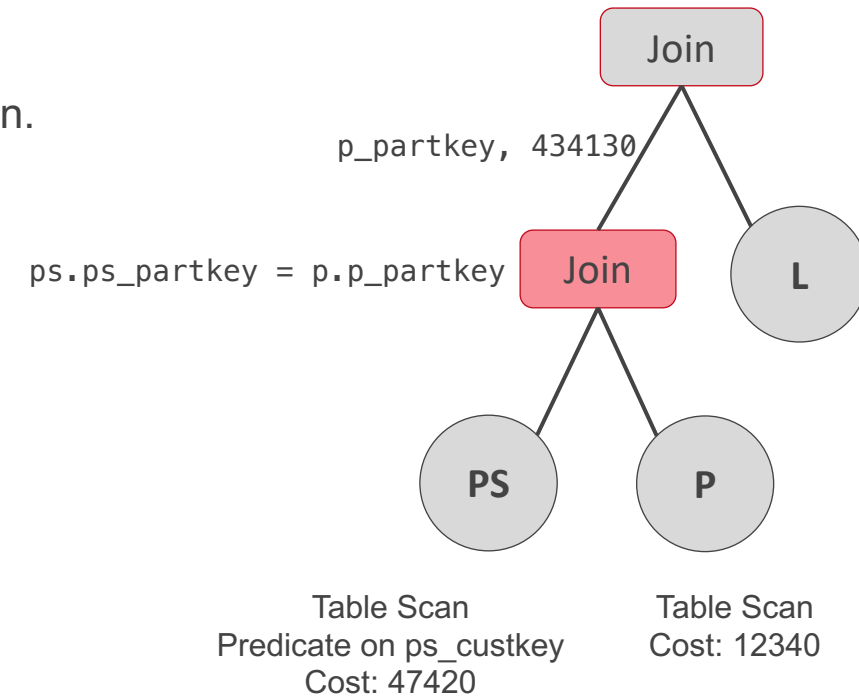
Cost: 77687
Ordered on p_partkey ✓



Cost: 4646681
No order ✗

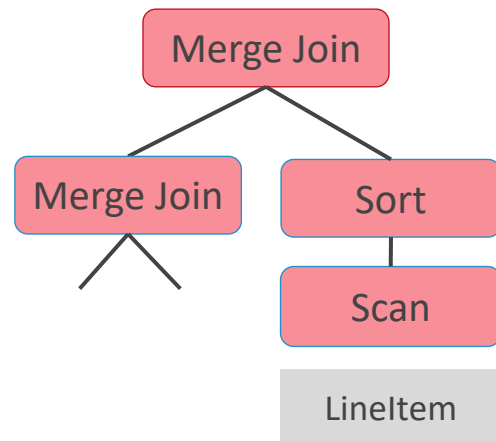


Cost: 49327420
No order ✗

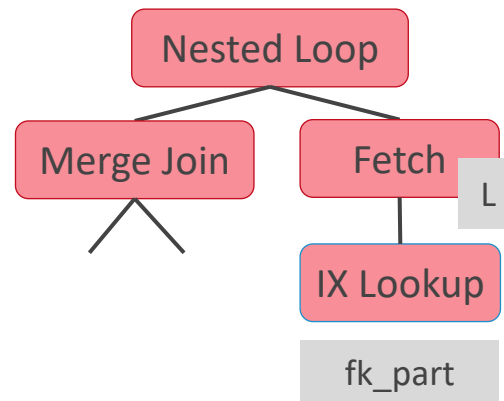


Example #3: Outer join

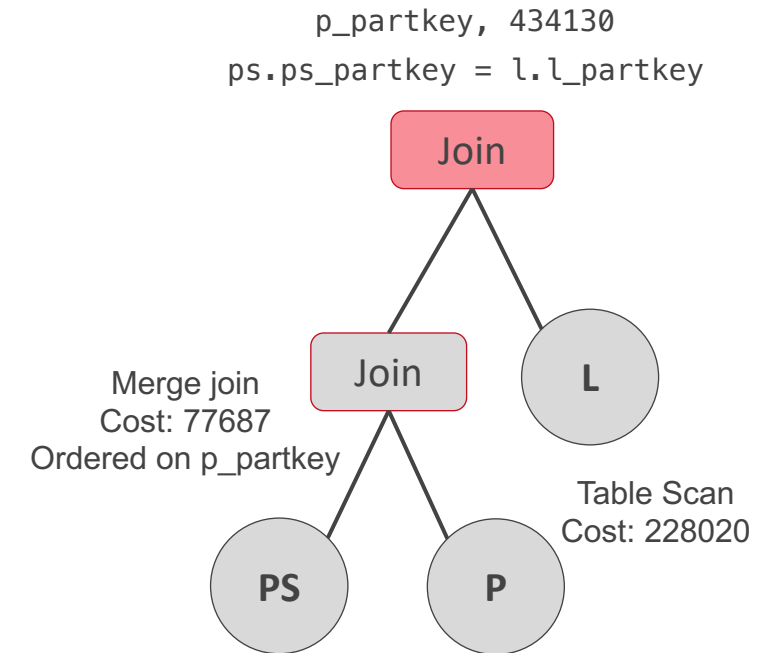
- No sort necessary for left input of merge join.
- Result of both join plans are sorted according to the interesting order.
- Only the merge join satisfies the interesting order because it is the cheapest plan.



Cost: 426623
Ordered on p_partkey



Cost: 100020643
Ordered on p_partkey



Example #3: ORDER BY

- Interesting order: `p.p_partkey, 434130`
- Result of outer join is sorted on `p.p_partkey`.
- No further sort necessary, can simply return result.

Implementation hints and helper functions

Cost estimation

- CardinalityEstimator
 - Passed to the `createPhysicalPlanGenerator` factory method.
 - Needs to be passed to some helper methods (see later slides).
- CostEstimator
 - Passed to the `createPhysicalPlanGenerator` factory method.
 - The cost estimator created in the previous exercise.
 - Use it to compute cost delta for interesting orders.
- PhysicalPlanCostUpdater
 - Helper to update operator and cumulative costs of an operator

Adding sorts and pruning plans

- `PhysicalPlanGeneratorUtils.addSortIfNecessary(OptimizerPlanOperator pop, RequestedOrder[] requestedOrder, int[] sortColumns, boolean[] direction);`
 - Check if a candidate plan pop is already sorted on requestedOrder.
 - If not, adds a SORT operator.

- `PhysicalPlanGeneratorUtils.prunePlans(OptimizerPlanOperator[] plans, InterestingOrder[] orders);`
 - Retain cheapest plan.
 - Retain cheapest plan for each interesting order if it doesn't exceed the cost delta.
 - You can just generate every possible candidate in each of the `build*` methods, and then call `prunePlans`.
 - The main difficulty is creating the interesting orders and passing them to subplans.

Joins

- `PhysicalPlanGeneratorUtils.isMergeJoinPossible(JoinPredicate predicate);`
 - Return true if the predicate is an equi join.
- `PhysicalPlanGeneratorUtils.createIndexNestedLoopJoinInner(...)`
 - Returns a subplan to access the inner table of an nested loop join using Index Lookup + Fetch.
 - Returns null if no index exists that can be used with the join predicate.
- `PhysicalPlanGeneratorUtils.addKeyColumnsAndAdaptPredicate(...)`
 - The left and right subplans of a join must return the projected columns and the columns used in the join predicate.
 - Updates the list of projected columns to include the join predicate columns.

Index lookup and visualization

- `PhysicalPlanGeneratorUtils.createIndexLookup()`
 - Creates an Index Lookup operator for equality and bounded range predicates.
- `OptimizerPlanVisitor.print(OptimizerPlanOperator operator)`
 - Prints the subplan below operator

Column classes

- Column
 - Describes a column.
 - Data type
 - Relation
 - Index of the column in the original table.
- ProducedColumn
 - Describes a column produced by a SELECT clause.
 - Name
 - Aggregation function, if any
 - Ascending or descending order if used in ORDER BY clause
 - Order rank, i.e., the position of the column in the ORDER by clause

Order classes

- Order
 - Enum describing the sort order: ASCENDING, DESCENDING, or NONE.
- RequestedOrder
 - Signals that a columns should be sorted.
 - Associates a Column instance with an Order enum.
 - Used as an array RequestedOrder[] to specify that a plan must/should be sorted.
 - If it is used in a merge join.
 - For a GROUP BY clause if it is compatible with an ORDER BY.
 - To specify interesting orders.
- InterestingOrder
 - RequestedOrder[] array.
 - Associates cost delta.

Exercise 9

Exercise 9

- Implement the public method of PhysicalPlanGenerator
 - `OptimizerPlanOperator generatePhysicalPlan(...)`
 - `OptimizerPlanOperator[] buildBestOrderByPlans(...)`
 - `OptimizerPlanOperator[] buildBestGroupByPlans(...)`
 - `OptimizerPlanOperator[] buildBestSubPlans(...)`
 - `OptimizerPlanOperator[] buildBestConcreteJoinPlans(...)`
 - `OptimizerPlanOperator[] buildBestRelationAccessPlans(...)`

Tests and points

Test name	Points	Description
testOptimizerJoinOf3Query1	1	3 joins, similar to example query #3, but not ORDER BY.
testOptimizerJoinOf3Query2	2	Example query #1: 3 joins
testOptimizerJoinOf3Query3	2	Example query #2: 3 joins
testOptimizerJoinOf3Query4	2	Example query #3: 3 joins, ORDER BY clause.
testOptimizerJoinOf4Query1	1	4 joins
testOptimizerJoinOf4WithGroupingQuery	1	4 joins, GROUP BY clause and HAVING clause.
testOptimizerJoinOf5WithGroupingAndSortQuery1	1	5 joins, GROUP BY clause, HAVING clause, ORDER BY clause, different sort directions.
testOptimizerJoinOf5WithGroupingAndSortQuery2	1	5 joins, GROUP BY clause, HAVING clause, ORDER BY clause, different sort directions, sort columns include aggregated column.
testOptimizerSimpleGroupingQuery	1	No Join, GROUP BY clause.
testOptimizerSimplePointQuery	1	No join, simple predicate in WHERE clause.
testOptimizerSimpleRangeQuery	1	No join, bounded range predicate in WHERE clause.

Further information

- P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. ACM SIGMOD 1979. <https://doi.org/10.1145/582095.582099>
- Seminal query optimization paper
- Combines join order optimization and access path selection in single pass

Committee Elections at TU Berlin



WHICH committees are being voted?

Faculty Boards

(Extended) Academic Senate

Board of Trustees

from **30. January - 01. February 2024**
10 - 15 h

WHY should I participate?

Exercise your **right** to vote to help shape our university!

➤ **Indirect influence** on the future of TUB

Working as an elected member of a committee:

➤ **Direct influence** on the future of TUB

- Network with interesting people
- Learn new perspectives
- Gain a wealth of experience of democratic opinion-forming processes

*Students have a right and duty to be involved in all committees!
Their involvement is decided by election*

So: TURN OUT AND VOTE!

further infos: www.tu.berlin/themen/wahlen

WHAT are they responsible for?

Committees decide on:

- **Content** and requirements of degree programs
- **Teaching** workload of academic staff
- **Appointments** of new professors
- and much more besides

HOW TO VOTE?

The elections shall be conducted as ballot box elections.

Postal voting can still be submitted until 24.01.24 via personal access in the TU portal.

<https://tuport.sap.tu-berlin.de/>

Keep in mind the letter delivery times.